

AD-A105 229

DEFENSE COMMUNICATIONS ENGINEERING CENTER RESTON VA
PROBLEMS WITH THE MULTITASKING FACILITIES IN THE ADA PROGRAMMIN--ETC(U)
MAY 81 S L ZUCKERMAN

F/6 9/2

UNCLASSIFIED

DCEC-TN-16-81

NL

1
2
3



END
DATE
FILMED
0 81
DTIC

1
LEVEL

12

TN 16-81



DEFENSE COMMUNICATIONS ENGINEERING CENTER

AD A105229

TECHNICAL NOTE NO. 16-81

PROBLEMS WITH THE MULTITASKING
FACILITIES IN THE ADA PROGRAMMING LANGUAGE

MAY 1981

DTIC
ELECTE
OCT 8 1981
A

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

81 10 7 035

DTIC FILE COPY

UNCLASSIFIED

April 1981

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

14 REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
REPORT NUMBER DCEC-TN-16-81	2. GOVT ACCESSION NO. AD-A105229	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Problems with the Multitasking Facilities in the Ada Programming Language		5. TYPE OF REPORT & PERIOD COVERED Technical Note
7. AUTHOR(s) Susan Lana/Zuckerman		6. PERFORMING ORG. REPORT NUMBER
9. PERFORMING ORGANIZATION NAME AND ADDRESS Defense Communications Engineering Center Computer Systems Division, R800 1860 Wiehle Ave., Reston, VA 22090		8. CONTRACT OR GRANT NUMBER(s)
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Communications Engineering Center Computer Systems Division, R800 1860 Wiehle Ave., Reston, VA 22090		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS N/A
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) N/A		12. REPORT DATE 11 May 1981
		13. NUMBER OF PAGES 18
		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES Review relevance 8 years from submission date		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Ada Programming Language Real-Time Applications Embedded Applications Multiprocessing Multitasking High Order Languages		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The Ada language was designed for military embedded computer applications, and therefore its multitasking facilities will be heavily utilized in DoD software. However, Ada's multitasking facilities do not provide capabilities equivalent to current multitasking facilities in existing languages. Ada's unconventional multitasking facilities raise serious questions. Problem areas are identified and discussed, and solutions are proposed. Because Ada's multitasking facilities are only a small part of the total language,		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED 407519
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Block # 20 Abstract

changes and improvements can be made without impacting the remainder of the language.

TECHNICAL NOTE NO. 16-81

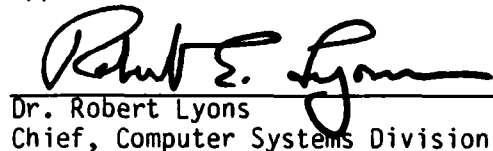
PROBLEMS WITH THE MULTITASKING FACILITIES
IN THE Ada PROGRAMMING LANGUAGE

MAY 11, 1981

Prepared by:


SUSAN LANA ZUCKERMAN

Approved for Publication:


Dr. Robert Lyons
Chief, Computer Systems Division

A

FOREWORD

The Defense Communications Engineering Center (DCEC) Technical Notes (TN's) are published to inform interested members of the defense community regarding technical activities of the Center, completed and in progress. They are intended to stimulate thinking and encourage information exchange; but they do not represent an approved position or policy of DCEC, and should not be used as authoritative guidance for related planning and/or further action.

Comments or technical inquiries concerning this document are welcome and should be directed to:

Director
Defense Communications Engineering Center
1860 Wiehle Avenue
Reston, Virginia 22090

EXECUTIVE SUMMARY

The Ada language was designed for military embedded computer applications including signal processing, weapon fire control, and data communications. As such, Ada's multitasking facilities will be heavily utilized in DoD software. The multitasking facilities in Ada depart from conventional methods and present a unique and original design, requiring the realtime programmer to think about tasking in a new and different way. The Ada multitasking facilities do not provide equivalent capabilities of current multitasking facilities in existing languages, and serious questions have been raised concerning Ada's new multitasking mechanisms. It remains to be proven by close analysis and extensive programming examples that these new concepts are both complete and safe for DoD multitasking applications.

Serious problem areas in the design of the multitasking facilities in Ada are identified in the text. For each language problem identified, reading references are given when applicable, the problem is discussed, and a proposed solution is recommended. The proposed changes would greatly improve the usability, reliability, and maintainability of realtime software written in Ada.

Fortunately, since the multitasking facilities of Ada are a small part of the language design, and in particular are orthogonal to the rest of the language, they may be changed and improved without impacting the majority of the design of the Ada language. It is suspected that the multitasking facilities of Ada were not reviewed by users and designers of realtime systems during the Ada design phase, and therefore it is recommended that the language be examined by realtime programmers and designers (e.g., designers of network software) to determine if the multitasking capabilities meet the realtime programming requirements. If Ada is found to be deficient in this area, it is recommended that multitasking facilities similar to other extant realtime languages, as discussed in this document, be incorporated into Ada.

TABLE OF CONTENTS

	<u>Page</u>
EXECUTIVE SUMMARY	2
I. INTRODUCTION	4
II. ADA LANGUAGE ISSUES	5
1. TASK ACTIVATION	5
2. ENTRY CALL CONSTRUCT	6
3. SUSPEND AND RESUME CAPABILITIES	7
4. TASK ACTIVATION PARAMETERS	8
5. TASK TERMINATION	8
6. DEFAULT TASK PRIORITY	9
7. TASK SYNCHRONIZATION LIMITATIONS	10
8. RENDEZVOUS AND PRIORITY	10
9. BLOCKING TASK COMMUNICATION	11
10. RENDEZVOUS QUEUING DESIGN PROBLEM	12
11. PASSING SHARED DATA AS PARAMETERS	13
12. RUNTIME SPECIFICATION OF TASK PRIORITY	14
13. DURATION TYPE IMPLEMENTATION	15
III. CONCLUSIONS	17
REFERENCES	18

I. INTRODUCTION

The Ada language was designed for military embedded computer applications, and therefore the multitasking facilities of Ada will be heavily utilized in DoD software. Yet problems remain with the multitasking facilities in Ada, and it is questionable if Ada's multitasking facilities are sufficient for realtime applications. The Ada multitasking facilities do not provide equivalent capabilities of current multitasking facilities in existing languages (e.g., SPL/I - the Navy's high order language for signal processing and realtime applications, or CHILL - the CCITT proposed standard high order language for telecommunications) or realtime operating systems (e.g., SDEX/M - the Navy's runtime executive for the CMS-2 language). In addition, some of the multitasking language constructs are vague and can be misunderstood. Language features and constructs should be distinct and well-defined, and should not occur as "side-effects" of other language features. Language constructs should also reflect the programmer's algorithmic thinking (hence, the IF THEN ELSE, DO WHILE, CASE constructs, etc.). While the majority of the Ada language follows this philosophy, the design of the multitasking portion of the language abandoned these precepts. Discussions of Ada's multitasking are often prefaced by the statement that multitasking is very difficult to understand. Yet there is nothing difficult or complex about multitasking, and in fact other languages provide multitasking facilities in well-defined, easy-to-understand language constructs.

Fortunately, since the multitasking facilities of Ada are a small part of the language design, and in particular are orthogonal to the rest of the language, they may be changed and improved without impacting the majority of the Ada language. It is suspected that the multitasking facilities of Ada were not reviewed by users and designers of realtime systems during the Ada design phase, and so it is recommended that the language be examined by realtime programmers and designers (e.g., designers of network software) to determine if the multitasking capabilities meet the realtime programming requirements. If Ada is found to be deficient in this area, it is recommended that multitasking facilities similar to CHILL, SPL/I or other realtime languages be incorporated into Ada.

II. ADA LANGUAGE ISSUES

The multitasking facilities in Ada depart from conventional methods and present a unique and original design. Specifically, the multitasking facilities are based upon a sole mechanism for task synchronization and communication called the "rendezvous". The rendezvous concept, while exciting, requires a new way of thinking about task communication. However, there are serious problems related to rendezvous, and it remains to be proven by extensive programming and analysis that rendezvous is both complete and safe for DoD multitasking applications.

Traditional methods of task communication and synchronization such as locks, counting semaphores, binary semaphores, and mailboxes are discussed in most texts on operating systems theory. One such text, useful for background reading, is listed as reference (7). Reference (1), the Ada Reference Manual, is the defining document for the Ada language, and includes the multitasking design of Ada.

Problem areas in the design of the multitasking facilities in Ada are discussed below. The following sections are organized by language issues. For each language issue identified, reading references are given (when applicable), a discussion of the problem is presented, and a proposed solution is recommended. The Ada language is defined in reference (1). SPL/I, the Navy's realtime programming language for signal processing and other applications, and CROS, the runtime support for SPL/I, are defined in references (2) and (3) respectively. CHILL, the CCITT standard communications language, is defined in reference (4). The RED language, which was the competitor of current Ada, is defined in reference (5). SDEX/M, the realtime executive used with the Navy's CMS-2 language, is defined in reference (6). The following references in the language issue discussions are to sections in the Ada Reference Manual, reference (1), unless otherwise noted.

1. TASK ACTIVATION

Task activation occurs as a side effect of other language constructs (e.g., calling a procedure or allocating an object), rather than via an explicit language construct, and can result in insidious task blocking effects.

1.1 References

See section 9.3.

1.2 Discussion

There are two ways to activate a task in Ada. One way is to call a procedure which has the desired task declared immediately within its declarative part. Such a task is said to be dependent upon the procedure.

The desired task is activated prior to the execution of the first statement of the procedure; however, the procedure cannot return until the newly activated task terminates, thus blocking the task calling the procedure. For example, if a procedure B has a task C declared within its declarative part and task A calls procedure B, then C will be activated but procedure B will not return (and so task A will be blocked) until task C terminates (possibly never).

The other way to activate a task in Ada is to declare an access (i.e., pointer) type which designates a task object and then allocate the task object. The allocator creates the task object and immediately effects its activation. The scope containing the access type declaration (e.g., procedure, task declaration) cannot be left until all task objects allocated and activated have terminated.

Task activation is implicit within the Ada language, hidden within other language constructs. Calling a procedure may cause a task to become blocked, possibly forever. Similarly, allocating objects which contain task object components may also cause a task to become blocked. Language constructs which can block task execution should be distinct and easy to distinguish from innocuous statements. In multitasking applications, programming errors often result in tasks being hung up and blocked. Debugging this situation is usually quite difficult and time consuming. Ada's method of task activation is restrictive and hidden from the programmer, making software design, debugging, and maintenance difficult; yet, Ada was designed to reduce software development and maintenance costs and improve reliability.

1.3 Proposed Solution

The language should provide explicit task activation syntax distinct from variable allocation and procedure invocation, such as

INITIATE task_object ;

(Note: The preliminary version of Ada contained an INITIATE statement; it was deleted from the current version of the language). This would result in improved program readability and clarity, and would facilitate program debugging and maintenance.

2. ENTRY CALL CONSTRUCT

The entry call construct in Ada looks like the procedure call construct.

2.1 References

See section 9.5.

2.2 Discussion

The syntax for an entry call is identical to the syntax for a procedure call. Entry calls and their corresponding accept statements can cause tasks to become blocked. Due to the identical syntax, it is hard to find entry calls when scanning code, making program debugging and maintenance difficult. See the discussion under the preceeding paragraph pertaining to distinct language constructs.

2.3 Proposed Solution

Make the entry call syntax distinct. For example, the following syntax could be used:

```
RENDEZVOUS entry_name (actual_parameter_list);
```

3. SUSPEND AND RESUME CAPABILITY

One cannot SUSPEND and RESUME tasks in Ada.

3.1 References

The SPL/I Common Realtime Operating System, the RED Language, and the SDEX/M Executive all contain examples of the SUSPEND and RESUME functions.

3.2 Discussion

Often in realtime applications a situation (i.e., crisis) arises which requires the suspension of current processing and a switch to a different "urgent" processing mode. After the crisis mode completes (i.e., crisis over), one needs to resume original processing. There is no way to do this in Ada, other than by relying solely on task priorities (see problems with task priorities in Ada, below). Other languages provide the capability to suspend and resume tasks.

3.3 Proposed Solution

Add SUSPEND and RESUME statements to Ada. The SUSPEND statement should cause the execution of the designated task to be suspended until it is continued via the RESUME statement.

4. TASK ACTIVATION PARAMETERS

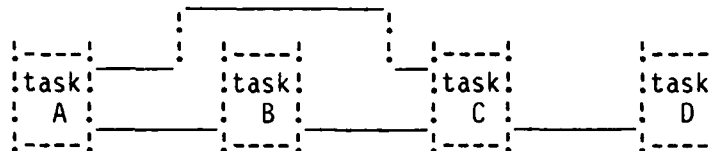
One cannot pass parameters to a task at activation.

4.1 References

The SPL/I, CHILL, and RED languages all permit passing parameters to tasks at activation time.

4.2 Discussion

Often in realtime applications, the output of one task is used as input to another task, whose output is in turn used as input to yet another task. The following diagram exemplifies the situation where tasks depend on input data provided by other tasks:



A clear and straightforward method is needed to specify input and output buffers to a task at activation time. In addition, a task may require processing parameters which change from one activation of the task to another.

4.3 Proposed Solution

Ada should have the capability to pass parameters to a task at activation similar to passing parameters to a procedure at invocation.

Example:

```
INITIATE task_object ( parameter_list );
```

5. TASK TERMINATION

One cannot unconditionally and immediately terminate a single task.

5.1 References

See sections 9.4 and 9.10.

5.2 Discussion

Often in realtime applications, a situation may occur which requires the termination of current processing and a switch to a different mode of processing. After the new mode completes, one may want to reactivate the previously terminated task(s). This is similar to the situation in paragraph 3 above, except that one needs to terminate tasks rather than suspend them. In Ada, normal task termination (reach end of task body), raising the task FAILURE exception, and using the TERMINATE option of a SELECT statement all cause the designated task to wait until all dependent tasks have terminated before terminating the designated task. The ABORT statement immediately terminates the designated task; however, it also immediately terminates all dependent tasks and this may not be desired.

5.3 Proposed Solution

Keep the existing task termination facilities in Ada but add a HALT statement that will unconditionally and immediately terminate only the designated task. Depending on Ada compiler implementation, users could be warned that the variables will disappear, and so dependent tasks should not access any variables inherited via scoping rules.

6. DEFAULT TASK PRIORITY

Task priority is not well-defined in Ada.

6.1 References

See section 9.8.

6.2 Discussion

Task priorities are essential to realtime applications. In Ada, task priority is specified via a compiler pragma. The priority pragma is not required and if it is not specified, a task's priority is not defined. Since task dispatching utilizes priority, a task which does not have a specified priority must be defined a default priority by the underlying operating system of any implementation. The various Ada implementations may treat the undefined case differently.

6.3 Proposed Solution

Define a default priority which should be known to the programmer. Most likely, the default priority should be the lowest priority possible.

7. TASK SYNCHRONIZATION LIMITATIONS

Task synchronization mechanisms are limited in Ada

7.1 References

See section 9.5.

7.2 Discussion

The only facility in Ada for task synchronization is the entry call. Entry calls are powerful tools for the programmer: however, they are somewhat restrictive in use and they have an associated runtime overhead. Semaphores are the classical synchronization mechanism in computer science. There are no semaphores predefined in Ada. Counting semaphores and binary semaphores could be built using entry calls defined in separate "service" tasks (e.g., define entries PSEMAPHORE and VSEMAPHORE in a task named COUNTING_SEMAPHORE, and have similar entries in a task named BINARY_SEMAPHORE). Implementing semaphores in this manner results in increased runtime overhead due to the additional dispatching and handling of the service tasks. It would be more efficient to have semaphores built directly into the language. In addition, having the individual users define and implement semaphores in this manner would result in various incompatible runtime libraries and would minimize software sharing.

7.3 Proposed Solution

Despite the problems with rendezvous (see below), it is recommended that the rendezvous facilities be left in Ada, but that additional synchronization mechanisms be added to the language (e.g., semaphores, regions, send and receive). By adding these features to the language, they could be implemented much more efficiently and would be available for common use. Realtime programmers would then have greater choice and flexibility in achieving task synchronization.

8. RENDEZVOUS AND PRIORITY

When two tasks attempting rendezvous have different priorities unexpected delays may occur.

8.1 References

See section 9.8.

8.2 Discussion

When the priority of a task doing an entry call differs from the task containing the entry, the rendezvous occurs via executing the called task at the higher priority. However, the calling task must wait for the called task to accept the rendezvous, which can result in unexpected delays. This is best explained by examples.

In the Ada language, rendezvous occurs as follows: If task A calls an entry defined in task B, task A and/or task B wait until A calls the entry and B accepts the entry. When both of these events occur, A is suspended while B executes the entry, and then A and B separate and continue their independent executions. Problems may occur when tasks A and B have different priorities. In particular, A may have a high (important) priority while B has a low (unimportant) priority. Then even though B may be ready to accept A's entry call, when A does the entry call A will be suspended until all tasks of priority less than A but greater than B have executed (because B cannot accept the entry call until it can run at its priority).

On the other hand, if the priority of task B is higher than the priority of task A, then task B will possibly waste CPU time by being swapped in and out and given unnecessary access to the CPU when it is waiting for a lower priority task A to do the entry call. This would occur if "service" tasks (e.g., similar to COUNTING SEMAPHORE described above) were assigned higher priorities than the tasks that used them.

8.3 Proposed Solution

This is a difficult problem that requires further study. One possibility is discussed here; however it remains to be shown that this is a safe solution. When a higher priority task does an entry call to a lower priority task, let the called task temporarily inherit the calling task's priority immediately (instead of only inheriting the calling task's priority during the actual rendezvous). Questions related to this solution remain to be studied. For example, consider the situation where several tasks do an entry call to a task and are queued. At what point should the called task's priority change? What happens if the called task is already involved in a rendezvous with a different task? Also, what happens if the called task's priority is increased, but it never performs the rendezvous (i.e., never executes an ACCEPT statement)? The unexpected task delay described above may be an inherent hazard of the rendezvous mechanism, and there may not be a solution to this problem.

9. BLOCKING TASK COMMUNICATION

A task cannot "SEND" a message to another task and continue execution prior to the "receipt" of the message.

9.1 References

The CHILL and RED languages both have SEND and RECEIVE mechanisms.

9.2 Discussion

This problem is a result of the design of rendezvous, which requires that both tasks wait for each other, rendezvous together, and then separate. Realtime tasks often need to signal or send data to another task without waiting for it to receive the signal or data. The best that can be done currently in Ada is for two tasks to exchange messages via a MAILBOX service task that has SEND and RECEIVE entries defined within it. This requires that the two tasks desiring to communicate rendezvous with the MAILBOX task rather than with each other; however, this alternative still assumes that the MAILBOX task is always available to rendezvous. There is no asynchronous, nonblocking communication path between tasks in Ada.

9.3 Proposed Solution

Add mailboxes and SEND and RECEIVE facilities to Ada, similar to the MAILBOX facility in the RED language.

10. RENDEZVOUS QUEUING DESIGN PROBLEM

Tasks waiting for rendezvous to occur are queued in *FIFO* (first-in, first-out) order rather than priority order, causing higher priority tasks to wait for lower priority tasks to run.

10.1 References

See section 9.5.

10.2 Discussion

If two or more tasks do an entry call to a task, the task selected for rendezvous first is the task that executed the entry call first (tasks are queued waiting for rendezvous in the order in which they attempted rendezvous). Hence, if a low priority task does an entry call before a higher priority task does the same entry call and they are both waiting for rendezvous, the low priority task is selected for rendezvous first.

Although other languages do not have the "rendezvous" concept, they do have synchronization mechanisms which cause tasks to wait for various events or conditions. Waiting tasks are often defined to be queued in priority order rather than FIFO order so that high priority tasks are not blocked by

lower priority ones. For example, SPL/I processes (tasks) waiting for semaphores or resources are queued in priority order; and CHILL processes may specify a queueing priority when they can conceivably become blocked.

10.3 Proposed Solution

Queue tasks waiting for rendezvous in task priority order, and for a given priority queue the tasks in FIFO order.

11. PASSING SHARED DATA AS PARAMETERS

The implementation for passing IN and INOUT parameters to a procedure (or rendezvous) is not defined; this impacts multitasking.

11.1 References

See section 6.2.

11.2 Discussion

The implementation of passing IN and INOUT parameters is not defined for arrays, records, and private type data. The Ada Reference Manual states that any program that relies on any one particular implementation is erroneous. This is fine for non-multitasking applications, as the decision to pass parameters by value or by reference simply affects the efficiency of the procedure and not the results of the procedure. However, in multitasking applications the implementation must be known by the programmer, and in fact controlled by the programmer.

Consider that if two or more tasks share data, each of them can read/modify that data at any time. If one task passes shared data to a procedure as an IN parameter, Ada leaves undefined whether the procedure always accesses the value the data had at the time of procedure invocation or whether the procedure accesses the current value of the data (possibly modified by other tasks). In multitasking applications, sometimes it is necessary that the former type of access occur (i.e., "call by value") and sometimes it is essential that the latter type of access occur (i.e., "call by reference"). As Ada is currently designed, it is unsafe to pass shared data as parameters to procedures, and in addition procedures cannot perform synchronization operations on their arguments. Similarly, the implementation is also not defined for parameters passed to a task via rendezvous. The consequences in this case are even worse because the rendezvous was designed to guarantee mutual exclusion!

11.3 Proposed Solution

The programmer must be given explicit control over the implementation of parameter passing. The programmer needs to choose between the following implementations: copy the value of the parameter at the time of procedure invocation, versus, pass a pointer to the data and always access the current value via the pointer. Given these requirements, there are several possible solutions. One solution is to define IN to be "call by value" (i.e., the parameter is copied into a local data area of the called procedure), and define a new keyword to designate "call by reference" (i.e., the parameter is always accessed directly via a pointer); similarly define INOUT to be "call by value" (i.e., the parameter is copied in and then copied out at procedure return), and add a new keyword to designate "call by reference". This would give the programmer direct control of parameter passing mechanisms.

12. RUNTIME SPECIFICATION OF TASK PRIORITY

A task's priority cannot be specified at runtime.

12.1 References

SPL/I allows a process's priority to, optionally, be specified at activation time via the ACTIVATE statement, thus overriding the default static priority (assigned at the time of software build). The CHILL language permits a process to specify a priority when it becomes blocked.

12.2 Discussion

In Ada, a task's priority is assigned statically in the specification part of a task declaration. For most situations, this method of specifying task priority is very useful, as in realtime applications a task's priority is normally assigned statically. However, it is sometimes necessary to override a task's priority when it is activated at runtime. For example, during "crisis" processing, it is often necessary to startup some tasks with increased or decreased priorities.

12.3 Proposed Solution

There should be an optional method of overriding a task's (statically specified) priority at the time the task is activated. Assuming the INITIATE statement described above is added to Ada, task priority should be specified as follows:

```
INITIATE task (parameter-list) WITH PRIORITY p ;
```

13. DURATION TYPE IMPLEMENTATION

The definition of the DURATION type used in task DELAY timing is poorly defined and too restrictive for 16-bit machines.

13.1 References

See sections 3.5.9 and 9.6.

13.2 Discussion

The DELAY statement is used to suspend the execution of the running task for (at least) the amount of time specified. The amount of delay time is specified via an argument of the predefined type DURATION. The Ada Reference Manual states that the predefined type DURATION must be a fixed-point type given in units of seconds, with the incremental delta to be left up to the individual compiler implementations of Ada. There is an added requirement that the DURATION type include both positive and negative values up to at least the number of seconds in a day (86,400 seconds). (Note that the DELAY of a negative value is defined to have no effect.)

Because the choice of an incremental delta is left up to individual compiler implementation, an indefinite number of programs written in Ada will not be transportable between different compilers, thus severely limiting software transportability. In addition, these Ada language requirements are very difficult to implement on 16-bit computers (e.g., the PDP-11). The largest integer value that can be represented in unsigned binary arithmetic using 16 bits is 65,536, while the largest integer value in two's complement notation is only 32,767. If a compiler implementation limited the timing increment to whole seconds (delta = 1), it would still be impossible to implement DELAY times using a single word. In fact, realtime applications require task DELAY time increments ranging from milliseconds (for signal processing and multilevel secure data communications applications) to seconds (for simple data communications protocols). For example, consider the requirements of the following data communications projects: (1) DCA's Communications Operating Systems/Network Front-End (COS/NFE) project requires a task delay time resolution of 20 milliseconds; (2) the Air Force AMPE project currently uses a task delay time resolution of one millisecond and is planning to convert to time resolutions of 100 microseconds; and (3) both the Mini R-TAC project and the ARPANET TIP software running on a PDP-11 under UNIX require task DELAY time resolutions of one second.

Sixteen bit computers such as the PDP-11 are heavily utilized in realtime applications, and in data communications applications in particular. In addition, most interval timer clocks in computers that would be used to implement DELAY times operate in time increments ranging from microseconds to milliseconds, but certainly not in increments of seconds. To define Ada language requirements that are quite difficult and expensive to implement on a most likely target computer for Ada seems unwise. The increased cost of implementing increments of seconds ranging from - 86,400 to +86,400 using

multiple words on 16-bit machines, in addition to the inefficiency of such an implementation, make the DELAY time requirements described in the Ada Reference Manual appear unreasonable. It is therefore recommended that the Ada DURATION type requirements be modified.

13.3 Proposed Solution

It is preferable that the DURATION type be fully specified to maximize software transportability, and that it be easy to implement on the majority of extant target computers. Given these aims there are two recommendations. First, the language requirement specifying that the range of time values in the DURATION type must include the number of seconds in a day should be deleted. This will avoid penalizing 16-bit target computers. Second, the type DURATION should consist of integer values of units of time, and either the units of the DURATION time should be changed to milliseconds (a common fine unit of resolution) or the units of time should be left up to individual implementations. This avoids the delta problem and would result in software that is transportable across compilers. (Note that if the definition of time units is left up to individual compiler implementations, Ada software will compile properly on alien compilers, but a delay time inconsistency may occur at runtime.)

III. CONCLUSIONS

Serious problems remain in the multitasking facilities of the Ada language which impact realtime applications such as data communications, signal processing, and fire control. Some of the problems discussed in this Technical Note can be circumvented through inefficient or unclear, "kludgy" programming. However, such convoluted programming methods obscure the algorithmic intent and result in greatly decreased software maintainability, reliability, and efficiency.

The Ada Reference Manual consists of over 208 pages (excluding index and some appendices). Modifying the tasking portion of Ada would basically involve rewriting chapter 9 of this document, which currently consists of only 16 pages. Because tasking is a small part of Ada and is orthogonal to the rest of the language, changes to the tasking facilities would not impact the remainder of Ada. The proposed changes would greatly improve the usability, reliability, and maintainability of realtime software written in Ada. In addition, it would simplify the implementation of the underlying operating system necessary to support multitasking in Ada.

Since multiple Ada compiler developments are currently underway, these multitasking issues need to be addressed as soon as possible. Most compiler development efforts are delaying the implementation of tasking to the end. If modifications to Ada's multitasking design are made in the near term, the impact on these compilers would be minimized.

REFERENCES

1. U.S. Department of Defense, "Reference Manual for the Ada Programming Language", July 1980.
2. Naval Research Laboratory, "SPL/I Language Reference Manual", January 1977.
3. Naval Research Laboratory, "User's Guide for the SPL/I Common Realtime Operating System (CROS)", April 1980.
4. The International Telegraph and Telephone Consultative Committee (CCITT), "The CHILL Language Definition", May 1980.
5. Intermetrics Inc., "Reference Manual for the RED Language", March 1979.
6. NAVELEX 0967-LP-598-2710, "Computer Program Performance Specification (CPPS) for Standard Executive for AN/UYK-20 and AN/AYK-14 Computers (SDEX/M)", 1 November 1979.
7. Tsichritzis and Bernstein, "Operating Systems", Academic Press, 1974.

DISTRIBUTION LIST

STANDARD:

R100 - 2	R200 - 1
R102/R103/R103R - 1	R300 - 1
R102M - 1	R400 - 1
R102T - 9 (8 for stock)	R500 - 1
R104 - 1	R700 - 1
R110 - 1	R800 - 1
R123 - 1 (Library)	NCS-TS - 1
R124A - 1 (for Archives)	101A - 1
	312 - 1
R102T - 12 (Unclassified/Unlimited Distribution)	
DCA-EUR - 2 (Defense Communications Agency European Area ATTN: Technical Library APO New York 09131)	
DCA-Pac - 3 (Commander Defense Communications Agency Pacific Area Wheeler AFB, HI 96854)	
DCA SW PAC - 1 (Commander, DCA - Southwest Pacific Region APO San Francisco 96274)	
DCA NW PAC - 1 (Commander, DCA - Northwest Pacific Region APO San Francisco 96328)	
DCA KOREA - 1 (Chief, DCA - Korea Field Office APO San Francisco 96301)	
DCA-Okinawa - 1 (Chief, DCA - Okinawa Field Office FPO Seattle 98773)	
DCA-Guam - 1 (Chief, DCA - Guam Field Office Box 141 NAVCAMS WESTPAC FPO San Francisco 96630)	
US NAV Shore EE PAC - 1 (U.S. Naval Shore Electronics Engineering Activity Pacific, Box 130, ATTN: Code 420 Pearl Harbor, HI 96860)	
1843 EE SQ - 1 (1843 EE Squadron, ATTN: EIEXM Hickam AFB, HI 96853)	
DCA FO ITALY - 1 (DCA Field Office Italy, Box 166 AFSOUTH (NATO), FPO New York 09524)	

(continued)

DISTRIBUTION LIST (cont'd)

USDCFO - (Unclassified/Unlimited Distribution)
(Chief, USDCFO/US NATO
APO New York 90667)

SPECIAL:

C400 - 1

C600 - 1

WSE-WIS - 1

NAVMAT 08Y - 1 (Department of Navy
Naval Material Command
ATTN: MAT 08Y
Washington, D.C. 20360)

ASN(RE&S) - 1 (Assistant Secretary of the Navy
Research, Engineering and Systems
ATTN: Mr. W. R. Smith
Pentagon Room # 5E779
Washington, D.C. 20350)